vwait — Process events until a variable is written

SYNOPSIS

vwait *varName*

DESCRIPTION

This command enters the Tcl event loop to process events, blocking the application if no events are ready. It continues processing events until some event handler sets the value of the global variable *varName*. Once *varName* has been set, the **vwait** command will return as soon as the event handler that modified *varName* completes. The *varName* argument is always interpreted as a variable name with respect to the global namespace, but can refer to any namespace's variables if the fully-qualified name is given.

In some cases the **vwait** command may not return immediately after *varName* is set. This happens if the event handler that sets *varName* does not complete immediately. For example, if an event handler sets *varName* and then itself calls **vwait** to wait for a different variable, then it may not return for a long time. During this time the top-level **vwait** is blocked waiting for the event handler to complete, so it cannot return either. (See the NESTED VWAITS BY EXAMPLE below.)

To be clear, *multiple* **vwait** *calls will nest and will not happen in parallel*. The outermost call to **vwait** will not return until all the inner ones do. It is recommended that code should never nest **vwait** calls (by avoiding putting them in event callbacks) but when that is not possible, care should be taken to add interlock variables to the code to prevent all reentrant calls to **vwait** that are not *strictly* necessary. Be aware that the synchronous modes of operation of some Tcl packages (e.g., **http**) use **vwait** internally; if using the event loop, it is best to use the asynchronous callback-based modes of operation of those packages where available.

EXAMPLES

Run the event-loop continually until some event calls **exit**. (You can use any variable not mentioned elsewhere, but the name *forever* reminds you at a glance of the intent.)

```
vwait forever
```

Wait five seconds for a connection to a server socket, otherwise close the socket and continue running the script:

```
# Initialise the state
after 5000 set state timeout
set server [socket -server accept 12345]
proc accept {args} {
    global state connectionInfo
    set state accepted
    set connectionInfo $args
}

# Wait for something to happen
vwait state

# Clean up events that could have happened
close $server
after cancel set state timeout

# Do something based on how the vwait finished...
switch $state {
    timeout {
        puts "no connection on port 12345"
    }
    accepted {
       puts "connection: $connectionInfo"
       puts [lindex $connectionInfo 0] "Hello there!"
    }
}
```

A command that will wait for some time delay by waiting for a namespace variable to be set. Includes an interlock to prevent

nested waits.

```
namespace eval example {
    variable v done
    proc wait {delay} {
        variable v
        if {$v ne "waiting"} {
            set v waiting
            after $delay [namespace code {set v done}]
            vwait [namespace which -variable v]
        }
        return $v
    }
}
```

When running inside a **coroutine**, an alternative to using **vwait** is to **yield** to an outer event loop and to get recommenced when the variable is set, or at an idle moment after that.

```
coroutine task apply {{} {
    # simulate [after 1000]
    after 1000 [info coroutine]
    yield

    # schedule the setting of a global variable, as normal
    after 2000 {set var 1}

    # simulate [vwait var]
    proc updatedVar {task args} {
        after idle $task
        trace remove variable ::var write "updatedVar $task"
    }
    trace add variable ::var write "updatedVar [info coroutine]"
    yield
}}
```

NESTED VWAITS BY EXAMPLE

This example demonstrates what can happen when the **vwait** command is nested. The script will never finish because the waiting for the *a* variable never finishes; that **vwait** command is still waiting for a script scheduled with **after** to complete, which just happens to be running an inner **vwait** (for *b*) even though the event that the outer **vwait** was waiting for (the setting of *a*) has occurred.

```
after 500 {
    puts "waiting for b"
    vwait b
    puts "b was set"
}
after 1000 {
    puts "setting a"
    set a 10
}
puts "waiting for a"
vwait a
puts "a was set"
puts "setting b"
set b 42
```

If you run the above code, you get this output:

```
waiting for a
waiting for b
setting a
```

The script will never print “a was set” until after it has printed “b was set” because of the nesting of **vwait** commands, and yet *b* will not be set until after the outer **vwait** returns, so the script has deadlocked. The only ways to avoid this are to either structure the overall program in continuation-passing style or to use **coroutine** to make the continuations implicit. The first of these options would be written as:

```
after 500 {
    puts "waiting for b"
    trace add variable b write {apply {args {
        global a b
        trace remove variable ::b write \
                [lrange [info level 0] 0 1]
        puts "b was set"
        set ::done ok
```

```
    }}}
}
after 1000 {
    puts "setting a"
    set a 10
}
puts "waiting for a"
trace add variable a write {apply {args {
    global a b
    trace remove variable a write [lrange [info level 0] 0 1]
    puts "a was set"
    puts "setting b"
    set b 42
}}}
vwait done
```

The second option, with **coroutine** and some helper procedures, is done like this:

```
# A coroutine-based wait-for-variable command
proc waitvar globalVar {
    trace add variable ::$globalVar write \
            [list apply {{v c args} {
        trace remove variable $v write \
                [lrange [info level 0] 0 3]
        after 0 $c
    }} ::$globalVar [info coroutine]]
    yield
}
# A coroutine-based wait-for-some-time command
proc waittime ms {
    after $ms [info coroutine]
    yield
}

coroutine task-1 eval {
    puts "waiting for a"
    waitvar a
    puts "a was set"
    puts "setting b"
    set b 42
}
coroutine task-2 eval {
    waittime 500
    puts "waiting for b"
    waitvar b
    puts "b was set"
    set done ok
}
coroutine task-3 eval {
    waittime 1000
    puts "setting a"
    set a 10
}
vwait done
```

SEE ALSO

global, **update**

KEYWORDS

asynchronous I/O, event, variable, wait